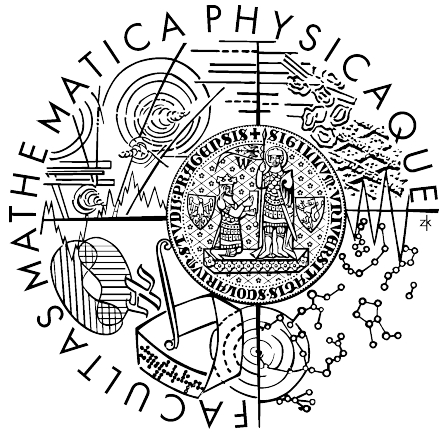
Univerzita Karlova v Prahe

Matematicko-fyzikálna fakulta



BAKALÁRSKA PRÁCA

Martin Molnár

Objektovo orientovaný príkazový interpret

Katedra softwarového inžinierstva

Vedúci bakalárskej práce: Mgr. Pavel Ježek

Študijný program Informatika, obor Obecná informatika

2008

Vyhlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním

V Prahe dňa 7. 8. 2008 Martin Molnár

Obsah

Názov práce: Objektovo orientovaný príkazový interpret
Autor: Martin Molnár

Katedra: Katedra softwarového inžinierstva

Vedúci bakalárskej práce: Mgr. Pavel Ježek

e-mail vedúceho: pavel.jezek@mff.cuni.cz

Abstrakt: Predložená práca sa zaoberá vnášaním objektovo orientovaného prístupu do programovania v príkazovom riadku. Študuje možnosti zľadenia konceptov OOP a dávkového spracovania a navrhuje vhodný kompromis medzi týmito prístupmi. Hlavným cieľom tejto práce je implementácia zvoleného riešenia v rodinách operačných systémov Unix a Microsoft Windows. Práca teda popisuje rozhodnutia súvisiace s implementáciou, ako aj používanie výslednej aplikácie.

Kľúčové slová: príkazový interpret, objektovo orientované programovanie, filozofia programovania

Title: Object-oriented Command Interpreter

Author: Martin Molnár

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Ježek

Supervisor's e-mail: pavel.jezek@mff.cuni.cz

Abstract: The goal of the proposed work is to explore possibilities of enriching command line interpreter programming by an object oriented approach. Author studies options of harmonizing concepts of the OOP and the batch processing and designs appropriate compromise between these two concepts. The main goal of the work is to create an implementation of the chosen design in the Unix and Microsoft Windows operating system families. This work describes decisions during the implementation and also include user manual of the resulting application.

Keywords: command interpreter, object-oriented programming, programming philosophy

Kapitola 1
Úvod

V dobe vzniku operačných systémov boli vytvorené príkazové interprety, ktorých základnou funkciou bolo spúšťanie úloh. Medzi časom, vznikom grafického rozhrania, sa vyvynuli iné spôsoby spúšťania programov, preto bola táto funkcia u príkazových interpretov oslabená. Tým sa však dostalo do popredia ich, oproti tlačítku štart a rôznym formulárom, silná stránka, a tou je možnosť programovaním skriptov automatizovať často alebo na viacerých počítačoch vykonávané postupy.

Ďalšou významnou zmenou v oblasti IT bol vznik objektovo orientovaného programovania, ktoré prináša programátorovi určitý komfort hlave v podobe intuitívnejšieho programovania.

Cieľom tejto práce a ročníkového projektu, na ktorý táto práca nadväzuje je sprístupniť objektovo orientované programovanie pre účely, na ktoré sa používa príkazový interpret. Výsledný shell má byť určený pre rodiny operačných systémov Unix® a Microsoft Windows®. Má byť dobre použiteľný a okrem iného poskytovať užívateľovi automatické doplňovanie identifikátorov a mien súborov. Tiež by mal podporovať prácu s COM objektami, pretože sa tým výrazne rozširujú možnosti jeho použitia.

Kapitola 2
Prehľad problematiky

Medzi spomínanými operačnými systémami existujú výrazné rozdiely. Ideálne by bolo, keby implementovaný shell (v ďalšom texte ho budeme označovať pracovným názvom `mosh`) umožňoval písať prenositeľný kód. Preklenúť tieto rozdiely do takej miery je však nepredstaviteľné. Shell by sa však mal pokúsiť čo najviac ich zakryť.

2.1 Operačné systémy

Rozdiely vo vonkajšej podobe operačných systémov

Rozdiely, ktoré sa priamo dotýkajú rozhrania, cez ktoré užívateľ vníma systém, je ťažké zakryť. Čo sa týka Unixu, táto rodina je značne rozvetvená a jednotlivé operačné systémy dosť rozdielne, všetky princípy súvisiace so shellom však majú rovnaké. Jednotlivé verzie operačného systému Windows sú v týchto princípoch tiež jednotné, zdedili ich od svojho predchodcu, operačného systému Microsoft DOS. Tieto rozdiely sú však značné medzi týmito rodinami. Na prvý pohľad je to najmä používanie iných oddelovačov adresárov v ceste a viac adresárových stromov v prípade Windows na rozdiel od jedného stromu v Unixe[1], to je ale len kozmetický rozdiel. Vážny problém tvoria koncepty, ktoré v druhej rodine nemajú svoj proťajšok. Takými sú napríklad systém prístupových práv (MS DOS a Windows 95, 98 a Me nemajú) a atribúty súborov (pôvodný Unix nemá). Obe rodiny však v poslednej dobe implementujú navzájom si podobný systém práv prostredníctvom tzv. Access Control Lists.

Rozdiely v rozhraní OS používanom shellom

Naopak rozdiely v princípoch používaných pri interakcii OS a shellu

1 Za zmienku tiež stojí fakt, že v Unixovom adresárovom strome sa nachádzajú aj ďalšie objekty, ako napríklad zariadenia, tiež v ňom zvyknú byť pripojené virtuálne súborové systémy, napr. procfs, debugfs. Pomenované pajpy a sockety existujú v Unixe aj vo Windows.

je možné úplne skryť. Ovplyvňujú len spôsob implementácie funkcí shellu.

Jeden z rozdielov je v predávaní parametrov vytváranému procesu. V Unixe je shell zodpovedný za rozdelenie príkazu na jednotlivé parametre a interpretáciu masiek ciest súborov v nich. Znamená to, že vznikajúci proces dostane ako argumenty cesty k súborom. Na rozdiel od toho vo Windows príkazový interpret odovzdá celý príkaz vznikajúcemu procesu, ktorý ho rozdelí na jednotlivé argumenty[2]. Taktiež je zodpovednosťou procesu, prostredníctvom k tomu určených systémových volaní, interpretovať masky ciest súborov. Dôsledkom toho je tiež fakt, že názvy súborov nemôžu obsahovať niektoré znaky[3].

Ďalším rozdielom je tiež spôsob vytvárania procesov. Vo Windows sa procesy vytvárajú volaním `CreateProcess` a všetky požadované vlastnosti a nastavenia sa odovzdávajú systému v argumentoch tohto volania. V Unixe najprv rodičovský proces zavolá `fork()`, operačný systém proces naklonuje, v rodičovskom procese vráti `fork` identifikačné číslo (PID) synovského procesu a v synovskom procese vráti hodnotu `0`. Synovský proces potom môže aplikovať požadované nastavenia štandardným spôsobom. Príkladom je zavretie deskriptorov súborov, ktoré musí v prístupe Windows rodičovský proces označovať za nededitelné špeciálnymi volaniami. Synovský proces v Unixe potom zavolá `exec(...)` ktorý načíta do pamäti nový obraz spustiteľného súboru.

Rozdiel je tiež v práci s konzolou. Vo Windows sa farba písma, pozícia kurzoru a ďalšie vlastnosti nastavujú špeciálnymi volaniami operačného systému. Na rozdiel od toho v Unixe existujú pre tieto účely špeciálne sekvencie riadiacich znakov, čo má za následok nežiadúce chovanie v prípade výpisu binárneho súboru na konzolu. Ostatné rozdiely v

2 keďže jazyk C bol vyvynutý pre Unix, má funkcia main argumenty prispôsobené Unixovému spôsobu ich odovzdávania. Preto vo Windows, kde je za rozdelenie na jednotlivé argumenty zodpovedný vznikajúci proces, toto rozdelenie vykováva C runtime pred zavolaním funkcie main.

3 v Unixe sú zakázané len 2 znaky v názve súboru, a to lomítko a nulový znak označujúci koniec reťazca v C.

operačných systémoch však nie sú konceptuálne a často sa dajú preklenúť použitím prenositeľných knižníc.

2.2 Príkazové interprety

V Unixe existuje niekoľko príkazových interpretov, napr. `bash`, `csh`, líšiace sa prevažne len syntaxou. Okrem toho sa na podobné účely sa niekedy používajú aj iné interpretované jazyky, ako napríklad `python` alebo `perl`. Vo Windows je klasickým a donedávna jediným príkazovým interpretom `cmd.exe`, v najnovších verziách Windows je však možné používať `Windows PowerShell`. Cieľom projektu je, aby implementovaný shell nahrádzal klasické príkazové interprety týchto systémov. Zamerajme sa preto na ich vlastnosti.

Funkcia príkazových interpretov

Základnou funkciou shellu je spúšťať procesy, so všetkými možnosťami, ktoré k tomu operačný systém ponúka. Samozrejmosťou sú argumenty pre spúšťaný proces, ale tiež aj nastavovanie environmentálnych premenných a presmerovanie ich vstupov a výstupov. V koncepcii týchto OS, v ktorej procesy spracúvajú svoj štandardný vstup a produkujú výstup, sa procesom na vstup a výstup prikladajú nielen súbory, ale aj výstup, resp. vstup iného procesu, čo dáva príkazovým interpretom možnosť riešiť komplexnejšie problémy.

Okrem toho shelli majú skriptovací aspekt, čo sa prejavuje cyklami a inými riadiacimi štruktúrami v ich jazykoch. Ďalšie vlastnosti shellu sú len priblížením užívateľovi, medzi ne patrí napr. automatické doplňovanie alebo aliasy v `bash`i.

Existujúce príkazové interprety

Tomuto popisu odpovedajú klasické nástroje, ktorými sú `bash` a `cmd.exe`. Prvý z nich je o niečo komplexnejší, čo je dôsledkom doby, v ktorej vznikal, užívateľmi a orientáciou daných operačných systémov.

Na druhú stranu `Windows Powershell` nezodpovedá tejto klasickej

definícii, pretože princíp programovania v ňom nie je postavený na spúšťaní procesov. Na rozdiel od toho používa pri dávkovom spracovaní hlavne tzv. commandlety, čo sú moduly implementované nie ako samostatné procesy, ale ako objekty v knižniciach systému. Existencia tohto protipríkladu nás nabáda hľadať obecnejšiu podstatu príkazových interpretov. V rozšírenej podobe by sme mohli shell definovať ako nástroj administrácie systému. Zatiaľčo v Unixe je administrácia postavená na programoch a editácii konfiguračných súborov, Windows poskytuje objektový prístup, napríklad vo forme WMI alebo .NET, čo je pre narastajúcu zložitosť operačných systémov veľká výhoda. Takáto koncepcia neposkytuje `cmd.exe` také pole realizácie, ako má `bash` v Unixe, čo je aj dôvod jeho slabého postavenia, naopak `Windows PowerShell` je práve preto veľmi silným nástrojom.

2.3 Windows PowerShell

Windows PowerShell implementuje z pohľadu príkazových interpretov inovatívny princíp objektovo-orientovaného programovania. Zostáva v ňom spájanie funkčných celkov pajpami (cez ktoré prúdia objekty), prispôsobuje ho však tak, že kód týchto celkov vykonáva vo svojom procese, na rozdiel od klasického spúšťania externých programov. Samotné commandlety — ako sa tieto funkčné celky nazývajú — sú implementované ako .NET objekty so špecifickým rozhraním, ktorým im PowerShell predáva parametry, ale hlavne postupne odovzdáva objekty na spracovanie. Commandlet môže počas spracovávania objektu vygenerovať ďalšie objekty, ktoré sa majú odovzdať jeho nasledovníkovi za pajpou. Na konci každého príkazu je implicitne cmdlet `Write-Host`, ktorý objekty generované posledným cmdletom vypisuje. PowerShell umožňuje zapojiť medzi cmdlety aj externé programy cez klasickú pajpu, v takom prípade musí objekty serializovať do textovej podoby.

Na podporu skriptovania poskytuje PowerShell premenné a štrukturované príkazy z procedurálneho programovania, napríklad vetvenia, cykly, funkcie a obsluhu výnimiek. Výhodou je tiež z .NET

frameworku prevzatý systém nastavovania bezpečnostných obmedzení zabezpečený elektronickým podpisovaním skriptov. Dôležitou vlastnosťou PowerShellu je, že nevytvára nové objekty pre úlohy administrácie systému, ale sprístupňuje už existujúcu bohatú a známu knižnicu objektov .NET.

Kapitola 3

Základné princípy objektovo orientovaného shellu

Cieľom projektu a tejto práce je vniesť objektovo orientované programovanie do shellu. Kľúčovou úlohou je zladiť ich princípy a vytvoriť vhodnú koncepciu. Nesprávne rozhodnutia v tejto fáze totiž môžu spôsobiť nevhodnosť výslednej aplikácie, alebo problémy s implementáciou niektorých jej vlastností.

3.1 Jednotlivé koncepcie

V úvode sa najprv zamyslíme nad samotnými koncepciami

Pajpa[4]

Jednou zo základných vlastností procesov v ich pôvodnej koncepcii bolo, že spracovávali súbor dát zvaný štandardný vstup a tak produkovali štandardný výstup. Shell má pritom možnosť nedať vytváranému procesu za vstup terminál, alebo súbor na disku, ale výstup iného procesu, čím je možné riešiť omnoho komplexnejšie problémy. Tento princíp je v protiklade s klasickým procedurálnym programovaním, ktoré používa prvky ako premenné alebo cykly. Hoci sa aj tieto prvky v shelloch vyskytujú, typickou vlastnosťou príkazových interpretov je dôraz kladený na dávkové spracovávanie.

Objekty

Hlavnou výhodou objektovo orientovaného programovania je intuitívnejší pohľad zo strany programátora, ale napríklad aj väčšia možnosť implementovať nápovedu vo forme automatického doplňovania

4 toto slovo je v podstate programátorský slang pochádzajúci z angličtiny. Nanešťastie ale v slovenčine nie je ustálený iný názov. Jeho možným ekvivalentom je rúra, ktoré pre zmenu ale pochádza z nemčiny. Najslovanskejšie alternatívy popisujúce objekty v angličtine popisované slovom pipe sú potrubie, prípadne rôzne druhy produktovodov (ropovod, vodovod, ...). S trochou nadsádzky by sa teda „pipe“ mohlo prekladať ako „dátovod“, prípadne v tejto práci „objektovod“.

identifikátorov.

Zladenie OOP a dávkového prístupu

Vyššie spomínané princípy však nie sú úplne zlúčiteľné, dokonca sú si istým spôsobom protichodné. Základom OOP je totiž individuálny prístup k objektom, napríklad vo forme virtuálnych metód. Objekty môžu na volanie tej istej funkcie reagovať odlišne, zatiaľčo v dávkovom prístupe sa najprv vybere modul, do ktorého sa potom spracovávané dáta posielajú. Kompromis je ťažké nájsť. Jedna možnosť je pre každý objekt zvlášť určovať, ktorá metóda sa zavolá, čo znamená potlačenie dávkového princípu. Druhá možnosť je zaviesť prísne typovanie a nepodporovať polymorfizmus. Nevýhodou tohto prístupu je prílišný formalizmus, ktorému sa užívateľ musí prispôsobiť, takéto jazyky napríklad vyžadujú deklarovanie premenných. Teoreticky existujú kompromisné riešenia, napríklad prístup, v ktorom sa určí polymorfne modul podľa prvého objektu prechádzajúceho pajpou. Nevýhodou je, že pajpou môžu tiecť rôzne objekty, preto to modul musí ošetrovať a takáto kompromisná koncepcia sa rozpadá.

Typovanosť v OOP

Existuje škála prístupov k tomu, aký význam sa v OOP kladie typom objektov. V kompilovaných jazykoch, napr. v C++, je nevyhnutné, aby mal každý objekt presne definovaný typ. Hlavnou výhodou tohto prístupu je rýchlosť prístupu k položkám, ale tiež odhalenie niektorých druhov chýb už pri kompilácii. Nevýhodou je nepružnosť v podobe nutnosti písania deklarácii typov, premenných a argumentov metód. Príkladom zmäkčenia tejto nevýhody je typ `Variant` vo `Visual Basicu`. Na druhú stranu interpretované jazyky zväčša typom neprikladajú veľký význam. Užívateľ v nich deklaruje triedy a ich metódy, do premenných však môže priraďovať bez obmedzení. Pri extrémnejšom prístupe, ako príklad slúži jazyk `ruby`, typy a triedy vôbec nehrajú rolu a každý objekt je potenciálne jedinečný. Vzhľadom k tomu, že shell je interaktívny a očakáva sa od neho úspora stlačených

kláves, deklarovanie typov sa preň nehodí.

3.2 Zvolený prístup

Fakt, že shell má byť objektovo-orientovaný znamená, že samotný proces shellu musí vykonávať funkcionalitu príkazov, na rozdiel od tradičných interpretov, v prípade ktorých bola táto ťarcha na procesoch, ktoré spúšťali. Dôvod je ten, že ak spojíme procesy cez pajpu, posielajú sa medzi nimi cez pajpu dátové prúdy. Ak by sme chceli medzi procesmi posielať objekty, musel by ich vysielajúci proces serializovať napr. do formy XML, a prímajúci proces späť rekonštruovať, čo by bolo výrazne neefektívne.

3.2.a Komponenty

Návrh `moshu` bol inšpirovaný konceptom vývoja software známym ako komponenty: Funkčné prvky sú moduly, ktoré majú vstupné a výstupné rozhrania[5] a celý príkaz je reprezentovaný niekoľkými modulmi pospájanými cez tieto rozhrania. Rozhranie je funkcia alebo sada funkcii s ich významom, pričom modul, ktorý je pripojený cez vstupné rozhranie volá tieto funkcie modul na druhom konci spojenia, ktorý má toto rozhranie ako výstupné. Tento prístup je natoľko intuitívny, že je lepšie ho objasniť na príklade, pre popis ktorého použijeme nasledovné rozhrania:

Cez rozhranie typu `IGet` s funkciou `Get` sa vracia 1 objekt.

Rozhranie `IPull` má funkcie `Begin`, `GetNext` a `AtEnd`, pomocou ktorých volajúci postupne získava od volaného postupnosť objektov.

Rozhranie `IExecute` má funkciu `Execute`, po zavolaní ktorej modul vykoná to, na čo bol naprogramovaný.

Napríklad keď zadá užívateľ príkaz `Echo "qwerty"`, shell vytvorí modul `Echo` s výstupným rozhraním `IExecute` a vstupným `IPull` a modul `ConstString` s výstupným rozhraním `IGet` a napojí vstup modulu `Echo` na výstup `ConstString`. Potom zavolá funkciu

5 známe viac pod anglickým názvom interface

`Execute`, ktorú má modul `Echo` vďaka rozhraniu `IExecute`. Modul `Echo` si pri vykonávaní `Execute` vypýta cez vstupné rozhranie objekty, od modulu `ConstString` dostane jeden a ten vypíše.

Jenen z typov rozhraní je IInStream, ktorým si volajúci pýta dátový prúd realizovaný pajpou, čo umožňuje ako moduly používať externé programy rovnakým spôsobom, ako v klasických shelloch.

3.2.b Skladanie modulov

Uvedený prístup je hierarchický v tom zmysle, že niekoľko modulov spojených cez rozhrania tiež splňuje definíciu modulu: Spolu plnia určitú funkciu a rozhraniami, ktoré nie sú použité medzi modulmi, ich možno spájať s inými modulmi. Preto mosh umožňuje priradiť nejakej skupine pospájaných modulov názov a potom ju používať v príkazoch.

3.2.c Základy syntaxe

Jedným zo základných faktorov pri návrhu syntaxe je zachovanie už zaužívanej syntaxe z klasických shellov, okrem iného kvôli zvyku užívateľov, ale aj kvôli úspore stlačených kláves. Syntax `moshu` je navrhnutá tak, že zápis názvov programov oddelených znakom pajpy spustí tieto programy tak, ako rovnaký príkaz v `bash`i alebo `cmd`. Obmedzením je konvencia, že názov programu musí začínať malým písmenom, čo je v Unixe zaužívané a vo Windows na veľkosti písmen nezáleží. Naopak veľkými písmenami začínajú názvy interných modulov. Ak táto konvencia nie je dodržaná, správna interpretácia názvu sa dá vynútiť pripojením výkričníka na začiatok názvu v prípade programu a analogicky znaku percenta v prípade modulu. Tento princíp rozlišovania prvým znakom rozšírime na ostatné typy tokenov:

typ tokenu	popis	príklady

externý program	prefix !; nie je nutný, ak názov začína malým písmenom	`sed`
		`notepad`
		`!X` [6]

6 Unixový grafický server X je jeden z mála programov, ktorý nerešpektuje unixovú konvenciu názvov programov z malých písmen

modul	prefix %; nie je nutný, ak názov začína veľkým písmenom	`Echo` `%Echo`
odkaz na bežiaci modul	prefix $	`$Variable`
reťazec	prefix @ alebo reťazec uzavrený do uvodzoviek	`"qwerty"` `@qwerty`
číslo	pozostáva z číslic prípade začína unárnym -	`10` `-10`
volba v argumente programu	začína znakom -	`--max-depth`

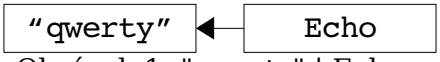
cesta	prefix ^; nie je nutný, ak cesta začína / ./ ~ alebo má začiatok typu c:	`/home`
		`./dir/file`
		`~/x`
		`c:\dir\file`
		`^.`

infixový binárny operátor	pozostáva zo znakov +-*:<=>	`+` `>>`

Ak má token obsahovať znak, ktorý v ňom normálne nie je dovolený, napríklad medzeru, ktorá slúži na oddeľovanie tokenov, je potrebné časť tokenu uzavrieť do uvodzoviek a to tak, aby prefix zostal pred prvou uvodzovkou. V opačnom prípade by bol totiž token interpretovaný ako textový reťazec.

Základom gramatiky je rozdelenie príkazu znakmi pajpy, takto získané časti nazveme úseky. V každom úseku sú medzerami oddelené moduly, pričom prvý má ako argumenty ostatné, tj. ostatné moduly majú výstupné rozhrania napojené na vstupné rozhrania prvého modulu. Prvé moduly každého pajpou oddeleného úseku sú potom spojené tak, že výstup ľavého je napojený na vstup pravého. Napríklad

```
"qwerty" | Echo
```

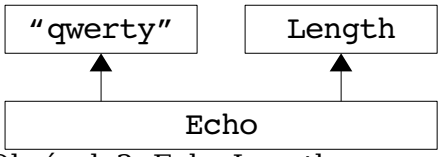


Obrázok 1: "qwerty" | Echo

```
Echo "qwerty“
```

sú ekvivalentné. V oboch prípadoch je `"qwerty"` napojený na `Echo`. Obídenie prioríť operátorov a iných pravidiel sa vynucuje hranatými zátvorkami. Preto

```
"qwerty" | Length | Echo
```

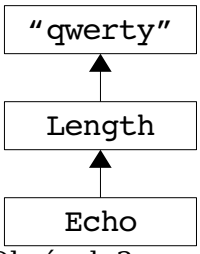


Obrázok 2: Echo Length "qwerty"

```
Echo [ Length "qwerty" ]
```

sú ekvivalentné, ale

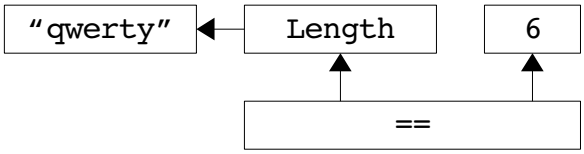
```
Echo Length "qwerty"
```



Obrázok 3: Echo [Length "qwerty"]

je od nich odlišný príkaz, pretože v tomto prípade je `Length` aj `"qwerty"` sú napojené na `Echo`[7]. V predchádzajúcich príkazoch je `"qwerty"` napojený na `Length` a ten je napojený na `Echo`. Infixné operátory majú nižšiu prioritu, ako pajpa, takže výraz

```
"qwerty" | Length == 6
```



Obrázok 4: "qwerty" | Length == 6

je korektný.

3.3 Vlákna

3.3.a Nutnosť používania vlákien

Fakt, že časť príkazu vykonáva shell a časť ním spustené procesy, prináša so sebou komplikácie. Užívateľ totiž môže zadať príkaz, v ktorom moduly posielajú dáta cez pajpu do spusteného procesu a z neho tečú dáta späť do modulov vykonávaných v shelli. Príkladom takéhoto príkazu je

```
1 | To 1000 | Echo | cat | Lines | Echo
```
[8]

Časti príkazu oddelené externými príkazmi budeme nazývať segmenty príkazu. Ak by existovala nejaká časová súvislosť medzi spracovávaním jednotlivých segmentov, napríklad tzv. dôsledné striedanie, hrozilo by zablokovanie. Preto musí byť spracovávanie segmentov časovo nezávislé. Principiálne sú dve možnosti riešenia:

---

7 Tento príkaz sa nepodarí preložiť, pretože modulu Length bude chýbať modul na vstupnom rozhraní, čo znamená, že nemá od ktorého modulu získať reťazec, ktorého dĺžku má vypočítať.

8 miesto cat môže byť použitý ľubovoľný Unixový príkaz, takisto moduly pred a za ním môžu byť rôzne. Príklady pre Windows sa hľadajú kvôli absencii štandardných programov na báze štandardného vstupu a výstupu ťažšie, ale principiálne je ten problém rovnaký.

1. spustiť vykonávanie kódu shellu spracovávajúceho každý segmenty v jeho vlastnom vlákne (z pohľadu operačného systému).
2. emulovať vlákna na aplikačnej úrovni

Druhá možnosť je omnoho komplikovanejšia. Jednotlivé procedúry by sa nemohli napríklad v prípade zápisu do pajpy blokovať, ale museli by uložiť svoj stav a vrátiť sa s chybovou hláškou určujúcou, že sa majú zavolať znova kvôli pokračovaniu. Shell by potom jednotlivé segmenty striedavo volal. Dalo by sa povedať, že koncept vlákien ako služby operačného systému je na riešenie tohto problému priamo stvorený.

3.3.b Pravidlá zvolené v návrhu

Zavedenie vlákien do ktoréhokoľvek softwarového projektu so sebou prináša špecifické problémy a nutnosť vytvoriť taký návrh, ktorý by týmto problémom predchádzal. Do návrhu sa preto dostalo niekoľko pravidiel, ktoré sa vláknami zaoberajú. Jedno z nich je, že všetok kód okrem modulov, tj. napr. čítanie a parsovanie príkazov, stavba modulov, atď ... vykonáva hlavné vlákno.

Ďalsím pravidlom je, že kód ľubovoľného modulu môže byť v každom čase vykonávaný len jedným vláknom. Ak má modul niekoľko výstupných rozhraní, musí túto podmienku chrániť pomocou mutexu zamykaného pri každom volaní funkcie z ľubovoľného rozhrania. Na druhú stranu veľká väčšina modulov má jedno výstupné rozhranie. V prípade takéhoto modulu M1 je modul M2, ktorý je cez jediné rozhranie napojený, jediná entita, ktorá môže volať metódy modulu M1, a keďže kód M2 induktívne spĺňuje túto podmienku, len jedno vlákno môže volať metódy modulu M1. Modul M1 preto nemusí používať žiadny mutex, pretože je chránený mutexom v module M2.

V tejto úvahe je skrytý jeden predpoklad a to, že na každé výstupné rozhranie môže byť napojený len jeden modul. Z hľadiska funkcionality je to obmedzenie; vysporiadame sa s ním zavedením tzv. zdieľacieho modulu pre každý typ rozhrania. Je to modul, ktorý má jedno vstupné rozhranie daného typu a neobmedzene mnoho výstupných rozhraní toho

istého typu[9]. Toto opatrenie nám rieši aj ďalší problém: Ak by na modul s výstupným rozhraním IPull boli pripojené 2 moduly a striedavo by sa pýtali na ďalší prvok, každý by dostal polovicu objektov. Metódy rozhrania totiž nemajú prostriedky na rozlišovanie toho, kto ich volá[10]. Riešenie, ktoré nám ponúkajú zdieľacie moduly, je opatriť zdieľací modul pre IPull cachovaním. Zobecnene preto definujeme zdieľací modul ako modul, ktorý sprostredkuje sémantiku rozhrania pre viac volajúcich.

3.3.c Prevencia zablokovania

Zavedenie zámkov prináša riziko deadlocku, s ktorým sa vysporiadame zavedením ďalšieho pravidla: Orientovaný graf modulov, v ktorom hrany zodpovedajú spojeniam modulov v smere od vstupného do výstupného rozhrania, bude musieť byť acyklický. Tým sme porušili jednu z Cuffmannových podmienok pre vznik deadlocku: čakanie do cyklu. Dôkaz predvedieme pre 2 vlákna:

Majme 2 vlákna, jedno bolo spustené volaním metódy modulu A_1, táto metóda volala metódu modulu A_2, ... a nech je vrchol zásobníka tohto vlákna funkcia modulu A_n. Toto vlákno preto drží zámky modulov A_1, A_2, ... A_n[11]. Analogicky, nech druhé vlákno drží zámky B_1, B_2, ... B_m. Pre spor budeme predpokladať, že nastalo čakanie do kruhu, to znamená, že druhé musí vlákno požadovať zámok, ktorý drží prvé a zároveň naopak. Preto modul A_n volá metódu modulu B_j ($1 \leq j \leq m$) a tá čaká na uvolnenie zámku B_j druhým vláknom a podobne B_m volá modul A_i ($1 \leq i \leq n$). Z toho ale plynie, že vrcholy grafu modulov A_i, ..., A_n, B_j, ..., B_m, A_i tvoria cyklus po hranách tvorených spojeniami z vstupného do výstupného rozhrania, čo máme zakázané. Spor.

Rozšírenie tohto dôkazu na ľubovoľný počet vlákien je triviálne.

9 na začiatku odstavca sme uviedli, že existuje skrytý predpoklad, bez ktorého je predchádzajúca úvaha o tom, že niektoré moduly nemusia byť chránené mutexom, chybná. Zdieľací modul je modul s viacerými výstupnými rozhraniami, preto musí obsahovať mutex, a tento mutex chráni modul, ktorého rozhranie zdieľa, pred vstupom viacerých vlákien do jeho kódu. Tým sme tento problém ošetrili a záver napadnutej úvahy ostáva v platnosti.

10 keby mali, každý modul by musel počítať s touto možnosťou, čo by komplikovalo kód

11 pokiaľ tieto moduly majú zámky. Formálne budeme predpokladať, že každý modul je chránený mutexom, teda aj tie, o ktorých sme ukázali, že to nepotrebujú.

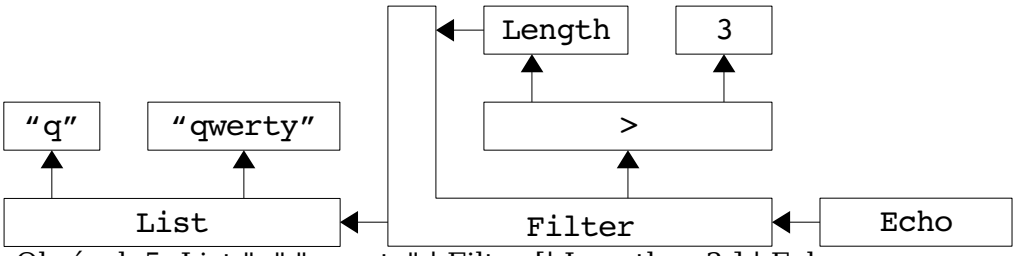
Nedostatok dôkazu je ale v tom, že sme predpokladali, že existujú len zámky chrániace vstup do modulov. Pritom je známe, že programátori shellových skriptov často potrebujú zámky na úrovni synchronizácie behov toho istého skriptu, častokrát si vypomáhajú atomickými operáciami typu zmazanie adresára. Po zahrnutí tohto faktu ale môžeme tvrdiť, že táto koncepcia nevnáša žiadnu náchylnosť na deadlock a teda že možnosť deadlocku je ekvivalentná tomu aká by bola, keby užívateľ napísal identický program v inom jazyku.

3.4 Syntaktické konštrukcie

3.4.a Spätný argument

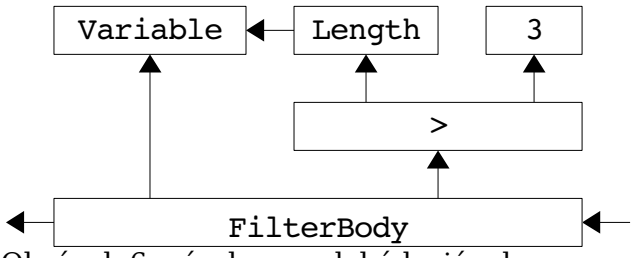
Keďže sa dáta prenášajú po spojeniach, pre zložitejšie úlohy je potrebné, aby bol modul napojený na svojho volajúceho. Príkladom je príkaz

```
List "q" "qwerty" | Filter [| Length > 3 ] | Echo
```



Obrázok 5: List "q" "qwerty" | Filter [| Length > 3] | Echo

ktorý vypíše tie z uvedených reťazcov, ktoré sú dlhšie ako 3 znaky. Modul `Filter` vyhodnocuje podmienku `Length > 3`, pričom modul `Length` je napojený späť do modulu `Filter`, odkiaľ získava reťazec, ktorého dĺžku má zistiť. Ako ukazuje príklad, spätný argument sa zapisuje do hranatých zátvoriek, pričom otváracia je nasledovaná znakom pajpy.



Obrázok 6: výsek z predchádzajúceho príkazu ukazujúci Filter rozdelený na 2 moduly

V podkapitole o vláknach sme stanovili pravidlo, že moduly musia tvoriť acyklický graf, čo

je v rozpore s uvedeným príkladom v ktorom vedie hrana z modulu `Filter` do modulu `>`, odtiaľ do modulu `Length`, a späť do modulu `Filter`. Tento problém je riešený tak, že `Filter` je zložený modul pozostávajúci z modulov `FilterBody` a `Variable`. Modul `FilterBody` si postupne berie objekty na spracovanie, každý najprv dosadí do premennej reprezentovanej modulom `Variable` a potom zavolá modul vyhodnocujúcu podmienku. Ten je priamo, alebo cez ďalšie moduly, napojený na modul `Variable`, ktorý mu na požiadanie vráti uložený objekt. Podmienka acyklicity sa vzťahuje na elementárne moduly, pretože tie vo svojom kóde obsahujú obsluhu mutexov. V implementácii je `Filter` šablóna zloženého modulu, takže je to v podstate len názov pre skupinu modulov, ktoré sa za behu chovajú individuálne.

3.4.b Direktívy

Ako sme už uviedli, `mosh` umožňuje napríklad dávať názvy zloženým modulom. Aby sa takéto príkazy odlíšili od seba navzájom a od klasických príkazov, ktoré majú za cieľ vykonať funkciu modulov, zavedieme direktívy. Sú to kľučové slová na začiatku riadkov, ktoré sa začínajú znakom „`:`“.

Základná funkcia shellu je spúšťať príkazy, k tomu slúži direktíva `:execute`, za ktorú sa píše výraz generujúci zložený modul s výstupným rozhraním `IExecute`. Táto direktíva je implicitná, preto sa `:execute` nemusí zapisovať. Shell ju spracuje tak, že z výrazu vytvorý zložený modul a zavolá metódu `Execute` jeho výstupného rozhrania.

Direktíva `:layout` má 2 argumenty a to názov novej šablóny a výraz, a spracováva sa tak, že sa z výrazu vytvorí šablóna zloženého modulu a priradí sa jej daný názov. Užívateľ potom môže používať v zápise ďalších príkazov miesto často sa opakujúceho výrazu tento názov, ktorý výrazu priradil.

Direktíva `:module` je podobná ako predchádzajúca s tým rozdielom, že sa vytvorí bežiaci modul, na ktorý sa odkazuje názvom s prefixom

„$“. Použitie názvu bežiaceho modulu v príkaze pripojí príslušné rozhrania na tento modul, zatiaľčo použitie direktívy :layout vytvára v každom príkaze nové moduly podľa popísanej šablóny.

Jedným z využití bežiaceho modulu je premenná, skrátene sa deklaruje pomocou direktívy `:var` nasledovanou typom a názvom. Týmto sa vytvorí bežiaci modul typu premenná a použitie jej mena v ďalších príkazoch napojí na daných miestach tento modul reprezentujúci premennú.

Direktíva `:include` otvorí skript, ktorý má v argumente a vykoná v ňom uvedené príkazy. Skript ukladá názvy definované pomocou `:layout` a `:module` v svojom kontexte, ktorý dosiahnutím konca skriptu zaniká. Na exportovanie názvu do rodičovského kontextu, tj. toho skriptu resp. užívateľského vstupu, ktorý volal `:include`, slúži direktíva `:export` s exportovaným názvom v argumente.

3.4.c Explicitné vstupy a výstupy

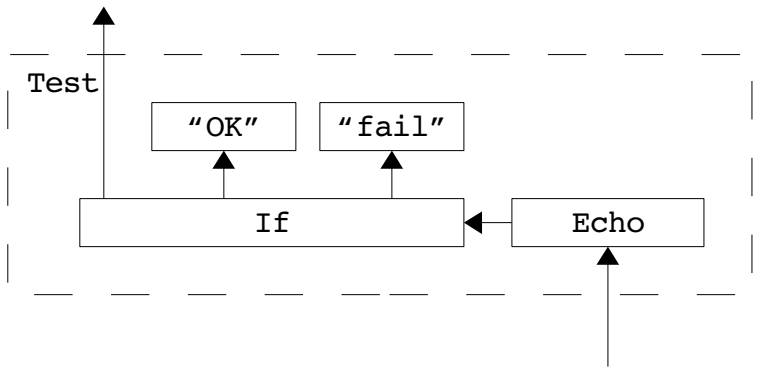
V zloženom module, napríklad definovanom direktívou :layout, má špeciálne postavenie tzv. posledný modul. Je to modul, ktorý je vzhľadom k relácii „byť napojený na“ najmenší. V zápise je to prvý modul za poslednou pajpou v príkaze. Predstavme si, že by sme príkaz spracovávali tak, že by sme jeho počiatočnú časť do bodu spracovávania považovali za zložený modul, a pripájaním nasledujúcich modulov by sme vytvárali zložitejší zložený modul. V tom prípade by bol posledný modul doposiaľ spracovanej časti práve ten, na ktorý máme pripájať ďalšie moduly. Preto budeme za implicitné výstupné rozhrania zloženého modulu považovať tie, ktoré vedú do jeho posledného modulu. Vstupné rozhrania zložený modul implicitne nemá.

Keď vytvárame šablónu zloženého modulu, hodí sa mať možnosť explicitne zadefinovať jeho vstupné alebo ďalšie výstupné rozhrania. Bez vstupných rozhraní by napríklad nebolo možné pomenovaný zložený modul použiť uprostred príkazu. Explicitne sa vstupné rozhranie vytvorí zápisom `[]<::` nasledovaným požadovaným typom rozhrania.

Napríklad v príkazoch

```
:layout Test [ If []<::Get "OK" "fail" | Echo ]

Test [ 0 == 0 ]
```

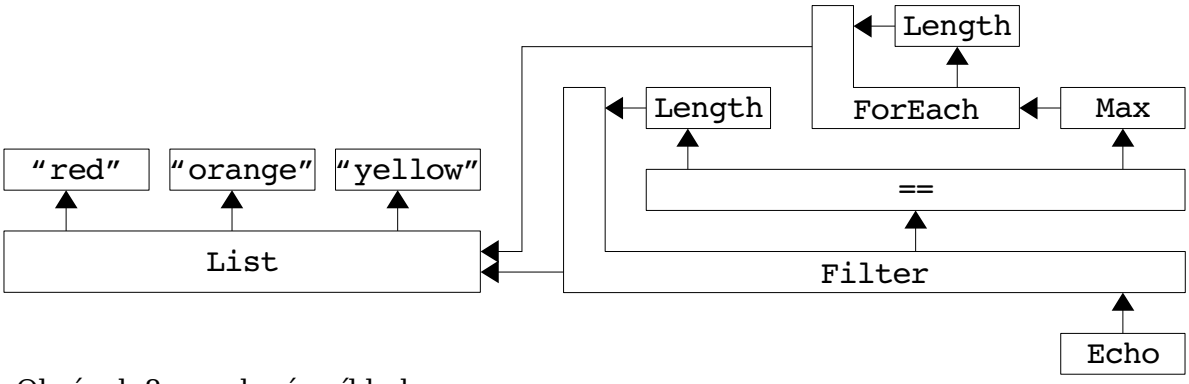


Obrázok 7: Novovytvorený modul Test

budú na modul If napojené reťazce `"OK"` a `"fail"` a modul `==`. Explicitné výstupné rozhranie sa zapisuje operátorom `::>` tiež s typom rozhrania.

Ďalším syntaktickým prvkom je možnosť pomenovať v príkaze modul, a jeho názov potom použiť, čím sa vytvorí medzi pomenovaným modulom a modulom, kde sa názov použil, spojenie (bez tejto možnosti, len pomocou pajpy a argumentov, by totiž mohli byť vytvárané len stromové štruktúry). Ilustruje to príklad[12]

```
[ List "red" "orange" "yellow" ]=::$X  \
 | Filter [| Length ==  \
             [ $X | ForEach [| Length ] | Max ]  \
          ] | Echo
```



Obrázok 8: uvedený príklad

12 napriek tomu, že je tento príkaz dosť zložitý, je asi najjednoduchší z rozumných použití ilustrovanej vlastnosti

ktorý vypíše tie z uvedených reťazcov, ktoré majú najväčšiu dĺžku. Na modul `List` je napojený nielen cez pajpu modul `Filter`, ale aj modul `ForEach`. Ten posiela modulu `Max` zoznam dĺžok spracovávaných pomocou modulu `Length`, modul `Max` z nich určí najväčšiu. Modul `Filter` postupne berie prvky zoznamu a testuje podmienku rovnosti a splňujúce prvky posiela modulu `Echo`. Pre úplnosť dodáme, že najväčšia dĺžka sa počíta pri každom porovnaní, zápis má teda kvadratickú zložitosť a je preto lepšie napr. uložiť dĺžku do premennej.

3.5 Porovnanie návrhu s koncepciou PowerShellu

Ako sme už uviedli, samotná snaha o objektovo-orientovaný príkazový interpret si vyžaduje presunutie vykonávania príkazu zo spúšťaných procesov do samotného shellu a ich zapúzdrenie do akýchsi modulov (commandletov). V tomto sú si PowerShell a `mosh` podobné. Rozdiel medzi nimi je, že PowerShell udržuje lineárnu štruktúru cmdletov spojených cez pajpy a ich prípadné parametre spracováva textovo podobne ako u klasických shellov. Na rozdiel od toho, `mosh` aj tieto argumenty reprezentuje modulmi a vytvára tým minimálne stromovú štruktúru modulov. Umožňuje mu to reprezentáciu zložitejších syntaktických prostriedkov, akými sú napríklad infixové operátory. Ďalší rozdiel je v tom, že zatiaľčo v `mosh`i sú moduly spojené tak, že priamo volajú funkcie susedov, PowerShell s každý cmdlet obhospodaruje sám, čo mu dáva väčšiu kontrolu. Je možné, že sa vďaka tomu vyhol problému asynchrónneho spracovania bez nutnosti zavedenia vlákien. Alternatívnou možnosťou bol totiž neblokovaný zápis do pajpy spusteného externého programu, ktorý je v takto centrálne riadenom spracovávaní jednoduchší.

Kapitola 4
Spôsob implementácie

Na implementáciu týchto koncepcii som zvolil jazyk C++, ktorý je podľa môjho názoru dobrým kompromisom medzi C a vyššími jazykmi (Java) pre tieto účely. Použili sa pritom viaceré knižnice z rodiny `boost`[13], pre prácu s vláknami `boost::threads`, pre spracovanie regulárnych výrazov `boost::regex`, a v rámci jednotlivých modulov napríklad aj `boost::filesystem`. Pre prácu s COM objektmi bola použitá knižnica `stlsoft`[14] a jej rozšírenie `vole`. Časť problémov počas implementácie bola technického charakteru alebo s priamočiarym riešením. Tie, ktorých riešenie vyžadovalo závažné rozhodnutia, prípadne bolo iným spôsobom netriviálne, popíšeme.

4.1 Objekty

Objekty v moshi sú dvoch typov a to klasické objekty a COM objekty. Klasické objekty sa skladajú z položiek vstavaných typov, ktorými sú `String`, `Integer`, `Boolean` a `Path`. Niektoré z dvojíc týchto typov majú medzi sebou zavedené implicitné konverzie. Ďalej existujú pseudotypy `Object`, na ktorý je konvertibilný každý typ, `Universal`, ktorý je konvertibilný na každý typ a typ `Null`, ktorý nie je konvertibilný so žiadnym typom. Tieto typy tvoria zväz a pre rozhodovanie o konvertibilite je každému typu priradené číslo z polozväzu prirodzených čísel usporiadaných reláciou deliteľnosti. Samotný objekt `moshu` sa skladá z jednej položky vstavaného typu, ktorá nie je pomenovaná, a z pomenovaných objektov `moshu`, ktoré môžu obsahovať ďalšie objekty.

Druhým typom objektov sú COM objekty. Shell používa na zisťovanie informácii o COM objektoch a na volanie ich metód, získavanie a nastavovanie hodnôt ich vlastností rozhranie `IDispatch`. Na manipuláciu s COM objektmi používa `mosh` knihovny `stlsoft` a `vole`.

13 http://www.boost.org
14 http://www.stlsoft.org

4.2 Rozhrania

Rozhrania sú implementované ako abstraktné triedy C++ obsahujúce čisto virtuálny popis funkcií, z ktorých rozhranie pozostáva. V prípade, že je jedno rozhranie potomkom druhého, potomok obsahuje definície metód predka, ktoré sa odvolálajú na nedefinované virtuálne metódy potomka. Príkladom je rozhranie `IGet`, ktoré je potomkom `IPull`. Zmyslom tohto vzťahu je, aby moduly, ktoré sú pripravené prijať postupnosť objektov cez `IPull`, mohli byť napojené na `IGet`, ktorý poskytuje jeden objekt. Rozhranie `IGet` obsahuje čisto virtuálnu metódu `Get` a implementáciu metód rozhrania `IPull`, konkrétne `Begin`, `GetNext` a `AtEnd`, ktoré vracajú 1-prvkový zoznam podľa sémantiky rozhrania `IPull`. Prvok zoznamu zistia volaním čisto virtuálnej metódy rozhrania `Get`, ktorej definícia je ponechaná na samotné moduly.

Ako uvidíme ešte niekoľkokrát, viaceré koncepty `moshu` majú samopopisovaciu schopnosť. Konkrétne u rozhraní, ktoré sú všetky potomkami triedy `Interface`, je budovaná paralelná štruktúra singletonov typu `InterfaceType`. Každý potomok `Interface` vracia jemu príslušiaci objekt typu `InterfaceType`, ktorý odpovedá na otázky o prevoditeľnosti jedného rozhrania na druhé (napr. už spomínaný prípad `IGet` a `IPull`).

Jednotlivé typy rozhraní môžu byť došpecifikované typom objektov, ktoré cezeň budú prúdiť. Napríklad modul `Length` bude na výstupnom rozhraní `IGet` vracať objekty typu `Integer`, preto nemôže byť napojený na vstup modulu `Delete`, ktorý požaduje objekty typu `Path`. `Length` ale môže byť napojený na modul `Echo`, ktorý požaduje na vstupe objekty typu `String`, pretože `Integer` je na `String` prevoditeľný. Poskytovanie informácie o type objektov tečúcich cez rozhranie by mohlo byť súčasťou rozhrania vo forme metódy, ale táto informácia nie je potrebná pri behu, ale už pri stavbe modulov, preto je poskytovaná ako súčasť samopopisovacej schopnosti modulov.

4.3 Moduly

Podľa už spomínanej koncepcie sú moduly akési funkčné jednotky, ktoré majú vstupné a výstupné rozhrania. Z hľadiska modulu mať výstupné rozhranie znamená ponúkať iným modulom možnosť volať metódy daného rozhrania. Preto je modul potomkom tried zodpovedajúcich jeho výstupným rozhraniam. Naopak mať vstupné rozhranie znamená pre modul možnosť volať metódy modulu, s ktorým je cez toto rozhranie spojený. Vstupné rozhrania sú teda implementované ako ukazovatele na triedu zodpovedajúcu rozhraniu. Ako sme už uviedli, potomkami tejto triedy sú práve moduly, ktoré majú toto rozhranie ako výstupné, a preto môžu byť práve tieto moduly do ukazovateľov dosadené.

Aj moduly majú samopopisovaciu schopnosť a to prostredníctvom objektov typu `ModuleProt`, ktoré zároveň slúžia na vytváranie modulov podľa návrhového vzoru Prototype. Takto poskytovaný popis pozostáva z názvu modulu, jeho vstupných a výstupných rozhraní a funkciou určujúcou typ objektov vracaných cez jednotlivé výstupné rozhrania. Na odlíšenie od prototypu modulu budeme samotný modul nazývať bežiaci modul a pojmom modul môžeme v abstraktnejšej rovine označovať kód reprezentovaný bežiacim modulom spolu s metainformáciami uchovávanými v prototype.

Po tom, čo je bežiaci modul zkonštruovaný, v prvej fáze svojho života, je modul spojovaný s inými modulmi v rámci budovania príkazu. K tomuto účelu má modul metódy `GetInterface`, ktorá vracia výstupné rozhranie[15] a `Connect`, ktorá pripojí na vstupné rozhranie iný modul. Do druhej fázi života — samotného vykonávania funkcionality — sa dostáva volaním `Close`, ktoré zároveň overí, či má modul napojené všetky vstupné rozhrania. Pre sprehľadnenie zápisu bola vytvorená šablóna `GenModule`, ktorá sa stará o všetky technické stránky modulu.

15 modul typicky vracia svoju adresu. Výnimkou sú zdielacie moduly, u ktorých je každé výstupné rozhranie reprezentované samostatným klientským proxy objektom, ktorý nesie informácie o stave, v ktorom sa nachádza dané rozhranie najmä v prípade rozhrania `IPull`, ktoré nie je bezstavové.

Medzi jej šablónové argumenty patrí výstupné rozhranie a typelist vstupných rozhraní, pričom je použitá technika, ktorú A. Andrescu nazýva v [1] „Class Generation with Typelists“. Táto šablóna je ale obmedzená a v neštandardných prípadoch, ako napríklad modul s viacerými výstupnými rozhraniami, sa tieto obslužné funkcie musia implementovať špeciálne.

4.4 Zložené moduly

Väčšina vstavaných modulov reprezentuje nedeliteľnú funkcionalitu a je implementovaná jednou triedou C++, takéto moduly budeme nazývať elementárne. Jedným z princípov programovania založeného na komponentách je možnosť skladať moduly do väčších funkčných celkov, akými sú zložené moduly. Funkciu prototypu zloženého modulu nesie trieda `Template`, ktorá obsahuje prototypy modulov, z ktorých zložený modul pozostáva a informácie o tom, medzi ktorými modulmi má viesť spojenie. Špeciálne tiež informácie o tom, z ktorých modulov má viesť spojenie mimo zložený modul, čo zodpovedá vstupným rozhraniam zloženého modulu a analogicky pre výstupné rozhrania. `Template` si uchováva prototypy modulov jednak preto, aby ako prototyp zloženého modulu vedela poskytovať metainformácie, a po druhé preto, že má na požiadanie vytvoriť bežiaci zložený modul. Ten je reprezentovaný triedou `Composite`, ktorá už obsahuje bežiace moduly, pospojované podľa informácii z `Template`.

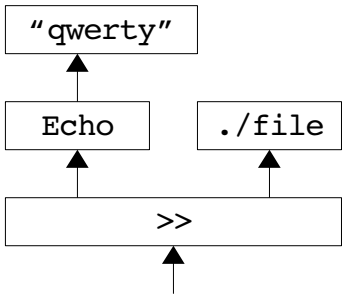
4.5 Spúšťanie procesov

K výkladu implementácie práce s procesmi nám poslúži zavedenie pojemu kontext. Je to trojica súborov: vstup, výstup, chybový výstup. Rozdeliť kontext znamená vytvoriť 2 kontexty - ľavý a pravý - tak, že vstup ľavého, výstup pravého a chybové výstupy oboch kontextov sú určené pôvodným kontextom a výstup ľavého a vstup pravého kontextu sú spojené novovytvorenou pajpou. Každé volanie modulu má ako argument kontext, v ktorom sa má vykonať, napríklad modul `Echo` vypisuje do súboru určeného kontextom, ktorý nemusí byť štandardný

výstup shellu.

Implementácia presmerovania vstupov a výstupov je potom priamočiara a objasníme ju na príklade

```
Echo "qwerty" >> ./file
```



Obrázok 9: Echo "qwerty" >> ./file

Tento príkaz je reprezentovaný 4 modulmi, ako to ilustruje obrázok 9. Po tom, čo shell vytvorí a spojí tieto moduly, zavolá `Execute` modulu `RedirOut` vo svojom štandardnom kontexte. Modul `RedirOut` zavolá `ConstPath`, ktorý mu vráti cestu `./file`, otvorí (prípadne vytvorí) tento súbor a vytvorí nový kontext, ktorý sa od pôvodného kontextu líši výstupom, ktorý je nastavený na deskriptor/handle novootvoreného súboru. V tomto novom kontexte zavolá `Execute` modulu `Echo`, ktorý bude zapisovať do novootvoreného súboru.

Zložitejšie je to so spájaním segmentov, tj. častí príkazov, ktoré vyžadujú časovo nezávislé spracovanie, ako je to opísané v kapitole 3.3. Na tieto účely slúži `IExecuteAsync`, ktoré rozdelí kontext novou pajpou, v ľavom podkontexte bude asynchrónne vykonávať svoju funkcionalitu a pravý podkontext vracia volajúcemu, v ktorom volajúci ďalej pracuje. Jediný modul, ktorý má ako výstupné rozhranie priamo `IExecuteAsync` je `Exec`, ktorý spúšťa proces. Ako príklad uvedieme

```
cat | Lines | Echo
```

ktorý je ekvivalentný príkazu `cat | cat` v bashi. Po zostavení je zavolaná metóda `Execute` modulu `Echo`, ten začne požadovať objekty od `Lines` a ten práve cez rozhranie `IExecuteAsync` dostane pravý podkontext, teda ten so vstupom z novovytvorenej pajpy. Modul `Exec` ešte pred vrátením nového kontextu samozrejme vytvorý proces `cat` v ľavom podkontexte, bude teda zapisovať do už spomínanej pajpy.

Ako sme už uviedli, jediný modul, ktorý má priamo výstupné rozhranie `IExecuteAsync` je `Exec.` Aby bolo možné použiť v segmente ľubovoľný modul ponúkajúci rozhranie `IExecute`, používame

dedičnosť rozhraní opísanú v kapitole 4.2. `IExecute` je potomkom `IExecuteAsync`, pričom obsahuje aj štandardnú implementáciu volania `ExecuteAsync`. Tá zavolá na príslušne rozdelenom kontexte v novovytvorenom vlákne metódu `Execute`, ktorú má definovanú samotný modul. Referencie na spustené vlákna a procesy sa ukladajú, aby shell mohol čakať na ich ukončenie.

4.6 Parser

V nasledujúcich dvoch podkapitolách popíšeme procedúry vedúce od prečítania príkazu cez jeho interpretáciu až po vybudovanie zloženého modulu a jeho beh.

`mosh` číta zo svojho vstupu riadok. Ak sa končí medzerou nasledovanou spätným lomítkom, spája sa s nasledujúcim riadkom; takto spojené riadky tvoria príkaz. Ten sa najprv rozdelí na tokeny, ktoré sú zadefinované regulárnymi výrazmi a na ich spracovanie sa používa knižnica `boost::regex`. Postupnosť tokenov sa interpretuje podľa bezkontextovej gramatiky, pričom sa využíva jej vlastnosť, že rozvíjaný neterminálny symbol a prvý terminál, na ktorý sa má neterminál rozvinúť jednoznačne určuje pravidlo, ktoré sa má použiť. Výsledkom parsovania je syntaktický strom zodpovedajúci príkazu.

Pôvodne sa uvažovalo o použití generátora zdrojového kódu `yacc/bison`. Nevyhovoval ale fakt, že `bison` je navrhnutý tak, že sám číta spracovávaný vstup, pritom v návrhu by viac vyhovovalo dať príkaz ako argument parsovacej funkcii. Vážnejším problémom bol však fakt, že vygenerovaný kód, a to ani pomocou `bisonc++` pre C++, nie je reentrantný, pričom mosh potrebuje niekoľko inštancii parsera v prípade direktívy `:include`.

4.7 Výstavba modulu

4.7.a graf

Zo syntaktického stromu sa vytvorí graf, ktorého vrcholy obsahujú

názvy modulov. Do tohto grafu sa vložia hrany identifikujúce požadované spojenia modulov, teda z ľavého do pravého operandu pajpy, z argumentu do modulu, ktorému náleží, ale napríklad aj hrany do spätných argumentov. Tento graf reprezentuje trieda `ModuleGraph`.

4.7.b obmedzujúce podmienky a ich riešenie

Shell potom musí nájsť takú konfiguráciu modulov, ktorá splňuje nasledujúce typy podmienok:

- Názov modulu je určený zodpovedajúcim vrcholom grafu (tzn. je potrebné zvoliť jeden z modulov s daným názvom).
- Spojnica modulov má určené jedno z rozhraní vstupného a jedno z rozhraní výstupného modulu tak, že výstupné rozhranie je konvertibilné na vstupné. Podobne v prípade explicitných vstupov a výstupov sa určuje rozhranie modulu kompatibilné s daným typom explicitného vstupu resp. výstupu.
- Na každé vstupné rozhranie modulu je napojený nejaký modul (okrem tých, ktoré to nevyžadujú).
- Do vstupných rozhraní modulu tečú objekty takých typov, aké je modul schopný spracovávať.

V algoritmickom popise riešenia tohto problému sa žiada použiť inštrukcia „vyber ten správny modul, potom to správne rozhranie, ...“, keďže ale počítač nie je nedetermistický, musíme to emulovať postupným vyberaním všetkých volieb a backtrackingom. Hľadanie takej konfigurácie je podobné riešeniu problému obmedzujúcich podmienok, avšak s tým rozdielom, že (v terminológii CSP) domény nie sú staticky dané pred začatím riešenia, ale sú utvárané počas riešenia. Všimnime si, že len pri prvých dvoch typoch vyberáme moduly resp. rozhrania, pri ostatných dvoch už len kontrolujeme konzistenciu predchádzajúcich výberov. Formálne je tento fakt implementovaný tak, že v prípade inkonzistencie nebudeme mať na výber žiadnu a v kladnom prípade jednu možnosť.

Pred začatím riešenia sú známe všetky podmienky, napríklad každému vrcholu náleží jedna podmienka prvého typu. Voľby pre danú podmienku ale určujú už uskutočnené voľby v rámci iných podmienok. Napríklad od toho, ktorý z modulov nazvaných `If` je vybraný pre splnenie prvej podmienky, môže závisieť nielen to, či daná voľba rozhraní spĺňuje druhú podmienku, ale aj to, koľko takých volieb existuje. Takýto vzťah medzi podmienkami budeme nazývať prerekvizita. Je zrejmé, že pri zahlbovaní sa vrámci backtrackingu musíme vyberať podmienky pre voľby v nejakom z topologických utriedení relácie prerekvizita. Avšak je dobré si uvedomiť, že použitie prerekvizít je omnoho voľnejšie v porovnaní s postupom, v ktorom by sme najprv vybrali voľby pre podmienky prvého typu, potom druhého atď. Graf modulov typicky nie je príliš prepojený, často je to dokonca strom, a uskutočnenie voľby pre prvý a potom druhý typ podmienky v nejakej jeho časti umožňuje overiť 3. a 4. typ pred rozhodovaním týkajúcim sa iných častí grafu, preto objavíme nutnosť backtrackingu skôr. Použitie prerekvizít nám navyše umožňuje spĺňovať najskôr podmienky s malým stupňom vetvenia, čo tiež optimalizuje backtracking. Napríklad výber voľby z jednej možnosti nevytvára riziko nutnosti backtrackingu a výber z dvoch možností môžeme odsúvať až do okamihu, keď budú všetky lepšie podmienky[16] prerekvizitne závisieť na našej voľbe.

Hľadanie riešenia riadi trieda `NondetSolver` a čiastočné riešenie je uchovávané v objekte triedy `Design`. Podmienky sú objekty typu `NondetChoice`, medzi ich základné funkcie patrí zistiť, koľko možností je možné zvoliť, aplikovať danú voľbu a vziať ju späť v prípade backtrakingu, aby bolo možné celé hľadanie prevádzať nad jednou kópiou `Design`.

4.7.c Vytvorenie šablóny a zloženého modulu

`Design`, ktorý obsahuje nájdené riešenie, skontroluje, či nie je na jedno výstupné rozhranie nejakého modulu pripojených viac modulov a v prípade, že je, vloží na príslušné miesto zdieľací modul. Zo štruktúry

16 tj. podmienky so stupňom vetvenia 1

designu sa viac-menej len zmenou reprezentácie vytvorí šablóna zloženého modulu. V prípade, že bolo vytvorenie modulu podnietené direktívou `:layout`, celý proces týmto končí, v prípade `:module` a implicitného `:execute` sa využije fakt, že šablóna slúži ako prototyp pre vytvorenie bežiaceho zloženého modulu.

Kapitola 5
Používanie aplikácie

Výsledkom implementácie je spustiteľná aplikácia, ktorá sprístupňuje užívateľovi objektovo orientovaný prístup pre úlohy shellu, a to vo forme jazyka navrhnutého v 3. kapitole. V tejto kapitole popíšeme spôsob jej používania. Aplikácia je určená pre Unix a Windows, pričom na oboch systémoch sa správa identicky, pokiaľ nie je v tejto kapitole uvedené inak. Zdrojové súbory, pre niektoré architektúry skompilovaná aplikácia a návod na inštaláciu a spustenie sa nachádza na priloženom CD.

5.1 Užívateľské rozhranie

5.1.a Automatické doplňovanie

Shell ponúka automatické doplňovanie mien modulov, spustiteľných programov a ciest k súborom. Funguje podobne ako v `bash`i; po stlačení tabulátoru shell zistí možnosti doplnenia tokenu pod kurzorom, ak je jedna, tak ju použije a ak je ich viac, možnosti vypíše.

5.1.b Chybové hlásenia

`mosh` má jednotný spôsob hlásenia chýb, ktorý ich rozdeľuje na nasledujúce úrovne:

úroveň	popis
`fatal_error`	chyba, ktorá znemožňuje ďalší beh shellu, napríklad chybný argument pri spustení
`error`	chyba, ktorá znemožňuje spracovanie príkazu, napr. syntaktická, ...
`exception`	chyba pri spracovávaní jedného objektu, napr. pri mazaní súboru. Ostatné objekty sú spracované normálne
`warning`	nastala legálna ale netypická situácia
`message`	hlásenie o tom, že sa niečo podarilo
`debug`	veľké množstvo hlásení vhodných na ladenie shellu

Implicitné je zobrazovanie hlásení úrovne `exception` a vyšších. Užívateľ môže túto úroveň zmeniť direktívou `:errors` nasledovanou úrovňou a názvom modulu, ktorého sa má zmena týkať, prípadne slova `all` pre všetky moduly. Napríklad

```
:errors debug all
```

zobrazí všetky hlásenia zo všetkých modulov. Toho je možné docieliť aj zadaním argumentu `--error-level debug` pri štarte `mosh`.

5.2 Programovanie v mosh

5.2.a Vstavané objekty a ich metódy

Užívateľ na písanie príkazov používa jazyk podľa syntaxe a sémantiky popísanej v 3. kapitole. Používa pritom tieto vstavané moduly:

modul	rozhrania[17]	popis
moduly pre prácu so zoznamami objektov		
`Count`	`+ IPull` `= IGet:Integer`	vracia počet objektov na vstupe
`First` `Last`	`+ IPull:X` `= IGet:X`	vracia prvý, resp. posledný z objektov na vstupe

`Nth`	`+ IPull:X`
	`+ IGet:Integer`
	`= IGet:X`
		vracia ten z objektov na vstupe, ktorého poradie je dané druhým vstupom

`Min` `Max`	`+ IPull:Integer` `= IGet:Integer`	vracia najmenší, resp. najväčší prvok zoznamu na vstupe
`Sum`	`+ IPull:Integer` `= IGet:Integer`	vracia súčet prvkov vstupu
`List`	`+ IPull:X` `= IPull:X`	zreťazí viac zoznamov do jedného

`To`	`+ IGet:Integer`
	`+ IGet:Integer`
	`= IPull:Integer`
		generuje postupnosť čísel medzi hranicami určenými vstupmi

práca s reťazcami		

17 vysvetlivky: symbolom + sú označené vstupné rozhrania, symbolom = výstupné. Za dvojbodkou sa uvádza typ objektov tečúcich cez rozhranie, ak na ňom záleží. Ak je tento typ X, môže byť ľubovoľný ale na všetkých rozhraniach rovnaký.

`Echo`	`+ IPull:String` `= IExecute`	vypíše reťazce zo zoznamu na vstupe

`Cat`	`+ IGet:String`
	`+ IGet:String`
	`= IGet:String`
		zreťazí vstupy

`CatNL`	`+ IGet:String` `= IGet:String`	pripojí na koniec reťazca znak nového riadku
`Length`	`+ IGet:String` `= IGet:Integer`	vráti dĺžku reťazca

`SplitBy`	`+ IGet:String`
	`+ IGet:String`
	`= IPull:String`
		rozdelí reťazec z prvého vstupu na podreťazce oddelené reťazcom z druhého vstupu

operácie s číslami a pravdivostnými hodnotami		

`+`
`-`
`*`
	`+ IGet:Integer`
	`+ IGet:Integer`
	`= IGet:Integer`
		infixové operátory

`Div`
`Mod`
	`+ IGet:Integer`
	`+ IGet:Integer`
	`= IGet:Integer`
		vracia celočíselný podiel a zbytok delenia

`<`
`<=`
`<>`
`>=`
`>`
	`+ IGet:Integer`
	`+ IGet:Integer`
	`= IGet:Boolean`
		infixové relačné operátory

`Not`	`+ IGet:Boolean` `= IGet:Boolean`	vracia negáciu

`And`
`Or`
	`+ IGet:Boolean`
	`+ IGet:Boolean`
	`= IGet:Boolean`
		vracia logický súčin resp. súčet, pričom používa skrátené vyhodnocovanie: ak prvá hodnota jednoznačne určuje výsledok, druhý argument sa nevyhodnocuje

riadiace moduly		

`If`	`+ IGet:Boolean`
	`+ IGet:X`
	`+ IGet:X`
	`= IGet:X`
		podľa hodnoty 1. vstupu vráti buď svoj 2. alebo 3. argument

`If`	`+ IGet:Boolean`
	`+ IExecute`
	`+ IExecute`
	`= IExecute`
		podľa hodnoty 1. vstupu zavolá vykonanie buď svojho 2. alebo svojho 3. argumentu. 3. argument môže byť vynechaný

`:=`	`+ ISet:X`
	`+ IGet:X`
	`= IExecute`
		vyhodnotí 2. vstup a cez rozhranie ISet ho 1. vstupu nastaví

`==`	`+ IGet`
	`+ IGet`
	`= IGet:Boolean`
		testuje rovnosť objektov

`===`	`+ IPull`
	`+ IPull`
	`= IGet:Boolean`
		testuje rovnosť zoznamov, tj. či obsahujú na rovnakých pozíciách rovnaké prvky

`<<`
`>>`
`:>>`
`>>>`
`:>>>`
	`+ IExecute`
	`+ IGet:Path`
	`= IExecute`
		Presmerovanie vstupov/výstupov
		`<<` presmeruje vstup
		`>>` a `>>>` presmerujú výstup
		`:>>` a `:>>>` presmerujú chybový výstup
		pritom `>>>` a `:>>>` zapisujú na koniec súboru

`StdIn`	`= IInStream`	vracia štandardný vstup
`Lines`	`+ IInStream` `= IPull:String`	číta otvorený súbor na vstupe a vracia zoznam riadkov
operácie nad systémom súborov		
`CurDir`	`= IGet:Path` `= ISet:Path`	vráti alebo nastaví (pomocou `:=`) aktuálny adresár
`CD`	`+ IGet:Path` `= IExecute`	nastaví aktuálny adresár

`IsDir`
`IsReg`
`IsSymLink`
	`+ IGet:Path`
	`= IGet:Boolean`
		testujú, po rade, či je argument cestou k adresáru, normálnemu súboru alebo symbolickému linku

`Exist`	`+ IGet:Path` `= IGet:Boolean`	testuje, či cesta odkazuje na existujúci objekt
`MkDir`	`+ IGet:Path` `= IExecute`	vytvorí adresár

`MoveTo`
`CopyTo`
	`+ IPull:Path`
	`+ IGet:Path`
	`= IExecute`
		presunie objekty z prvého argumentu do adresára daného druhým argumentom

`Rename`	`+ IGet:Path`
	`+ IGet:Path`
	`= IExecute`
		premenuje objekt súborového systému

`Delete`	`+ IPull:Path` `= IExecute`	zmaže objekty na vstupe

`Size`	`+ IGet:Path` `= IGet:Integer`	vráti veľkosť súboru
`Expand`	`+ IGet:Path` `= IPull:Path`	argument spracuje ako wildcard a vracia cesty, ktoré mu vyhovujú
ostatné		
`ToInteger`	`+ IGet:String` `= IGet:Integer`	interpretuje vstupný reťazec ako číslo
`ToPath`	`+ IGet:String` `= IGet:Path`	interpretuje vstupný reťazec ako cestu
`System`	`+ IGet:String` `= IGet:Boolean`	vracia TRUE na ak je vstup „unix“ alebo „linux“ na unixových systémoch, alebo „windows“ alebo „win“ na Windows
`ScriptArgs`	`= IPull:String`	vracia argumenty spusteného skriptu
`Exit`	`= IExecute`	ukončí shell

5.2.b COM objekty

Na vytvorenie COM objektu slúži modul `COMObj`, ktorý v argumente prijímá názov, pod ktorým je daná trieda registrovaná v systéme. Napríklad

```
COMObj "Excel.Application"
```

vracia novovytvorenú inštanciu aplikácie Excel. Na COM objekte potom môže užívateľ volať metódy a získavať a nastavovať vlastnosti, pričom používa operátor `->` a zátvorky, do ktorých uvádza prípadné argumenty volania. Pri volaní metód a získavaní hodnôt vlastností je podporované len vracanie objektov existujúcich v shelli, tj. reťazcov, čísel, pravdivostných hodnôt a iných COM objektov.

Ako príklad práce s COM objektami uvedieme príkazy

```
:var COM $Excel

:var COM $Book

:var COM $Sheet

$Excel := COMObj "Excel.Application"

$Excel -> Visible() := 1
```

```
$Book := $Excel -> Workbooks() -> Add()
$Sheet := $Book -> ActiveSheet()
$Sheet -> Range("A1") -> Value() := "qwerty"
```

ktoré postupne vytvoria COM objekt aplikáciu Excel, nastavením jej vlastnosti `Visible` ju zviditeľnia, vytvoria v nej nový zošit a do bunky A1 zapíšu text „qwerty“.

5.3 Príklady

Pre ilustráciu používania shellu uvedieme ešte niekoľko príkladov. Príkaz

```
./* | Filter [| IsDir ] | Count | Echo
```

vypíše počet podpriečinkov aktuálneho adresára.

```
1 | To 10 | Sum | Echo
```

vypíše 55 (súčet čísel od 1 do 10)

```
List "riadok1" "riadok2" | Echo >> ./dva_riadky
```

zapíše dva riadky do súboru ./dva_riadky.

```
./dva_riadky | Lines | Echo
```

vypíše obsah súboru ./dva_riadky.

```
./"áéí“ | Mkdir
Mkdir ./"áéí“
```

vytvorí adresár ./áéí (oba zápisy sú ekvivalentné).

```
CD ./testdir
```

zmení aktuálny adresár.

```
./testfile | MoveTo ./testdir
```

presunie súbor ./testfile do adresára ./testdir, analogicky modul `CopyTo` kopíruje.

```
./*.tmp | Filter [| IsReg ] | Delete
```

zmaže súbory (regulárne tj. nie adresáre, symbolické linky, ...) končiace príponou tmp v aktuálnom adresári.

`find ^. -type "f" | xargs "grep" "qwerty"`

spustí `find . -type f | xargs grep qwerty` (v unixových systémoch).

`notepad`

spustí notepad (vo Windows).

`Exit`

ukončí `mosh`.

Kapitola 6
Záver

Koncepcia modulov je podľa môjho názoru zaujímavá, prepracovaná a vhodná na toto použitie. Jej implementácia - vytvorená aplikácia spľňa základné funkcie nástrojov `bash` a `cmd`, pritom sa z časti drží jednoduchého zápisu ich príkazov. Navyše prináša objektový prístup a podporu COM objektov.

Výhodou moshu je, že je dostupný pre obe rodiny operačných systémov, naopak výhodou všetkých zaužívaných nástrojov a to nielen vo sfére príkazových interpretov je, že sú užívateľovi známe a je na ne zvyknutý. Ich používanie tiež dávalo podnety na ich rozšírenia, napríklad aliasy v `bash`i. Veľmi dôležitá vlastnosť nástrojov určených na tieto účely je ich stabilita, ktorá sa do detailov buduje tiež až používaním, pri ktorom sa objavujú aj veľmi špecifické chyby.

K skvalitneniu projektu by prispela väčšia škála funkcionality, ktorú možno pridávať jednoducho pridávaním nových modulov. Inšpiráciou k tomu bude tiež až každodenné používanie.

Prílohy

7.1 Formálny popis gramatiky

```
QUOTED            ((\'([^\']*)\')|(\"([^\"]*)\"))+
STRINGPART        ([^][[:space:]|+*<>=:-()]|{QUOTED})+

whiteSpace        [[:blank:]\n]+
comment           #[^\n]*\n
moduleName        \${STRINGPART}
layoutName        (%{STRINGPART})|([[:upper:]][[:alnum:]]+)
progExecName      (!{STRINGPART})|([[:lower:]][[:alnum:]]+)
redirectionOp     <<|:?>?>>
relationOp        [<>=:]+
binaryOp          [-+*<>=:]+
constString       (@{STRINGPART})|{QUOTED}
constInteger      [-+]?[[:digit:]]+
constPath         (/|\\|\./|\.\\|~/|~\\|\^|[[:alpha:]]:){STRINGPART}+
constOption       -([[:alpha:]]|-|{STRINGPART})+

Name ::= moduleName
       | layoutName
       | progExecName

Constant ::= constString
           | constInteger
           | constPath
           | constOption

Iface ::= layoutName
ClosedExpr ::= "[" Expr "]"
             | ClosedExpr "::" Iface
             | ClosedExpr "::>" Iface
             | ClosedExpr "=::" moduleName
             | "[]<::" Iface

MArgument ::= ClosedExpr
            | Name
            | Constant
```

```
MCalling ::= Name
           | MCalling MArgument
           | MCalling "[|" Expr "]"

Simple ::= MCalling
         | Constant
         | ClosedExpr
         | ClosedList

List ::= Simple
       | List "," Simple

ClosedList ::= "()"
             | "(" List ")"

Expr3 ::= Simple
        | Expr3 "->" layoutName ClosedList

Expr2 ::= Expr3
        | Expr2 redirectionOp Expr3

Expr1 ::= Expr2
        | Expr1 "|" Expr2

Expr9 ::= Expr1
        | Expr9 binaryOp Expr1

Expr ::= Expr9
       | Expr relationOp Expr9

objectType :== layoutName
Line ::= Expr
       | ":execute" Expr
       | ":export" Name
       | ":include" constPath
       | ":layout" layoutName Expr
       | ":module" moduleName Expr
       | ":var" objectType moduleName
       | ":errors" Name Name
```

Literatúra

[1] Alexandrescu A.: *„Modern C++ Design: Generic Programming and Design Patterns Applied“*, Addison Wesley, 2001

[2] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1998